Data structures

Šimon Brecher

September 2022

Contents

1. Introduction

This is my "zápočtový program na Programování 2".

This project contains three data structures: binary tree, heap and dictionary.

This project is written in C# and is meant to be used by adding to another C# project as source code.

1.1 How to use

1. Download the source code on one of these URL adresses:
   https://github.com/simonbrecher/data_structures/tree/no_test
   https://github.com/simonbrecher/data_structures/tree/main
2. Add the source code to your C# project.
3. Now you can import the classes. (or run the file Program.cs to run tests)

2. Data structures

2.1 Binary tree

This structure an AVL tree. It is meant to be used as a set. It can only contain **int**.

2.1.1 Operations

This subsection is to list all of the operations and specify how the more complicated are implemented.

Name	Input	Return	Description
Create		BinaryTree	Create empty binary tree (constructor)
Add	int key		Add key, if it does not exist
Remove	int key		Remove key, if it exists
Has	int key	bool	Return true, if key exists
Deepcopy		BinaryTree	Return a deepcopy of itself
Enumerate		int[]	Return array of all keys sorted ascending

Balance If a vertex's children have depths different by 2, it can be corrected by switching few vertices. Then we need to balance all of the parent vertices as well. There are two different cases, depending on how high is depth of the subtrees A, B, C and D. (and two more symmetrical cases)

```
    y            x
   / \          / \
  x   C   ->   A   y
 / \              / \
A   B            B   C
```

```
A = (d + 1) ; B = (d) or (d + 1) ; C == (d)
```

```
    z              y
   / \            / \
  x   D          x   z
 / \      ->    / \ / \
A   y          A  B C  D
   / \
  B   C
```

```
A = (d) ; B = (d - 1) or (d) ; C = (d - 1) or (d) ; D = (d)
```

Remove Balancing tree after removing a vertex depends on, how many children it has. If it has only one child, ignore green arrow.

Green arrow: copy only key
Orange arrow: copy key and both pointers to children
Find green arrow by going once left, then right as long as possible.

2.1.2 Data representation

This data structure consists of one class: BinaryTree. An instance of this class represents one vertex in the binary tree (it can be considered root). This is, so we do not need two classes for this data structure (tree and vertex), but it makes some operations harder. For example a BinaryTree instance can not

directly switch itself with its child, because it can not access the pointers in its parent or user variables. Instead it uses shallowcopy.

Usually a vertex has both children and a key. If vertex should not have a child, it points to BinaryTree.EMPTY, which is a static instance of BinaryTree representing non-existing vertex. BinaryTree.EMPTY has itself as both children. Vertex does not have a key, only if it is BinaryTree.EMPTY or if it represents root of an empty tree.

Size is number of keys (keys are always non-duplicate). Depth is the longest path from a vertex to root. If tree does not have any keys, it has depth := -1.

2.2 Heap

This structure is a binary minimal heap. In every "vertex" is saved one key (can be duplicate) and user can also save there one **int?** value.

2.2.1 Operations

This subsection is to list all operations and specify how the more complicated are implemented.

Name	Input	Return	Description
Create		Heap	Create empty Heap (constructor)
Union	Heap heap1 Heap heap2	Heap	Create Heap as union (constructor)
Add	int key int? data=null	HeapVertex	Add key:data and return its object
Remove	HeapVertex vertex		Remove key:data (must belong to this Heap)
GetMin		HeapVertex	Return smallest key
RemoveMin		HeapVertex	Remove and return smallest key
Change	HeapVertex vertex int key		Change key
Union	Heap heap		Add all from other Heap to itself
Deepcopy		Heap	Return deepcopy of itself
EnumerateKey		int[]	Return all keys in saved order
EnumerateData		int[]?	Return all data in saved order

Union Union is implemented by Deepcopy and repeated Add.

2.2.2 Data representation

This data structure consists of two classes: Heap and HeapVertex. Heap represents whole structure and HeapVertex represents pair key:data. HeapVertex is needed to remove and change key or data values.

HeapVertex saves: key, data, position in heap and if it has been removed from the heap.

Vertices are saved as an array. First vertex is saved at index 0.

2.3 Dictionary

This structure is a dictionary. It can only contain **int** as both key and value.

2.3.1 Hashing

Hashing function only hashes **uint** $\rightarrow$ **uint**. Because of that, **uint** keys are decreased by **Int32.MinValue**.

Every dictionary has two random odd **uint** variables: A and B.

F(x, i) is hash function, where x is hashed value and i is number of iterations.
F(x, i) := (G(x) + i * H(x)) % Capacity
G(x) := A * x
H(x) := B * x (increased by 1, if it would be even)

2.3.2 Operations

This subsection is to list all operations and specify how the more complicated are implemented.

Name	Input	Return	Description
Create		Dictionary	Create empty dictionary (constructor)
Add	int key, int value		Add new key:value pair, if key does not exist Change value, if key exists
Has	int key	bool	Return true, if key exists
Get	int key		Return value, if key exists Return null, if key does not exist
Remove	int key	int?	Remove key:pair value, if key exists Return value, if key existed Return null if key did not exist

Add Find first position (hash) with IsUsed[h] = false. Set: IsUsed[h] = true and WasUsed[h] = true.

Has / Get Find first position (hash) with WasUsed[h] = false. (Or wanted key value.)

Remove Find first position (hash) with WasUsed[h] = false. (Or wanted key value.) Set IsUsed[h] = false.

2.3.3 Data representation

This data structure consists of one class: Dictionary.

All data are saved in 4 arrays. Every position can save only one key:value pair:

Name	Type	Description
Keys	int[]	key (if IsUsed[i])
Values	int[]	value (if IsUsed[i])
IsUsed	bool[]	true, if there is key:value pair in this cell true values are counted by Size
WasUsed	bool[]	true, if there was key:value pair in this cell true values are counted by SearchSize

Important static constants. Dictionary will be rehashed, if any of its rules are broken. Capacity of Dictionary is always a power of 2:

Name	Type	Value	Description
MIN_CAPACITY	uint	2^3	
MAX_CAPACITY	uint	2^{31}	
MAX_SEARCH_SIZE	float	0.85	Maximal relative SearchSize
MAX_SIZE	float	0.7	Maximal relative Size
MIN_SIZE	float	0.25	Minimal relative Size

3. Automatic tests

3.1 Purpose

Purpose of the automatic tests is not to test the code in development. (I used manual testing, which was at the time much easier) Automatic tests are used to test the function after finishing development.

Automatic tests test, that all variables are in correct format (to each other) and that all outputs are correct for the real data structure and fake data structure.

3.2 Classes

Every data structure has (in most cases) one of each class. They have similar format, but most of the code is different, which is why I implemented each of the classes separately, without inheritance. (I mean corresponding classes between structures.)

Name	Description
TestXXX	Fake data structure Is slow, or from C# library
TestWrapper	Contains both real and fake data structure Has same methods as both structures Calls both structures with the same parameters Compares return values
Tester	Checks if TestWrapper(s) have correct values
Tests	Specifies which TestWrapper methods will be called Calls Tester Runs every test x1000

3.3 Expected output

```
---------------------------------------------------------------------
You may only use the Microsoft .NET Core Debugger (vsdbg) with
Visual Studio Code, Visual Studio or Visual Studio for Mac software
to help you develop and test your applications.
---------------------------------------------------------------------
Testing BinaryTree
    Add ASC, [-100, 100):                          Success (x1000)
    Add, [-100, 100) x200:                         Success (x1000)
    Add -> Remove, [-100, 100) x150 x300:          Success (x1000)
    Add + Remove, [-100, 100) x500:                Success (x1000)
    Add + Remove, [0, 10) x10000:                  Success (x1000)
Ended testing BinaryTree without any errors.
Testing Heap
    Add -> RemoveMin, [-100, 100) x200:                                  Success (x1000)
    Add -> Remove -> RemoveMin, [-100, 100) x200 50% :                   Success (x1000)
    Add -> Change -> RemoveMin, [-100, 100) x200 :                       Success (x1000)
    Add -> Union -> RemoveMin, [-100, 100) x100 :                        Success (x1000)
    Add -> Union (constructor) -> RemoveMin, [-100, 100) x70 :           Success (x1000)
Ended testing Heap without any errors.
Testing Dictionary
    Add ASC, [-100, 100):                          Success (x1000)
    Add, [-100, 100) x400:                         Success (x1000)
    Add -> Remove, [-100, 100) x200 x200:          Success (x1000)
    Add + Remove, [-100, 100) x800:                Success (x1000)
Ended testing Dictionary without any errors.
```

3.4 Found mistakes

BinaryTree In BinaryTree.Balance() i wrote Size instead of Depth. This allowed the Binary tree to have child depths different by 2.

BinaryTree I changed shallowcopy to deepcopy in private constructor after few weeks (was not private at the time). This caused infinite loop.

Dictionary Dictionary.Remove() returns value of removed key. The dictionary can be rehashed, if there is too little key:value pairs saved after removing. I counted hash before, but returned the value after rehashing. This caused different return value.

Dictionary I compared SearchSize to Size, instead of Capacity. Because of this, the Dictionary would never rehash in case of too high SearchSize.

Dictionary Dictionary.Add() ended when IsUsed[i] == false, instead of WasUsed[i] == false. This allowed duplicate key values.

4. File structure

4.1 Without automatic tests

Namespace	Class	Directory
DataStructures	BinaryTree	/src/
DataStructures	Heap	/src/
DataStructures	HeapVertice	/src/
DataStructures	Dictionary	/src/
DataStructures	DataStructuresException	/src/

4.2 With automatic tests

Namespace	Class	Directory
DataStructures	BinaryTree	/BinaryTree/
DataStructures	Heap	/Heap/
DataStructures	HeapVertice	/Heap/
DataStructures	Dictionary	/Dictionary/
DataStructures	DataStructuresException	/Exceptions/
Test	TestException	/Exceptions/
TestBinaryTree	TestBinaryTree	/BinaryTree/
TestBinaryTree	TestWrapper	/BinaryTree/
TestBinaryTree	Tester	/BinaryTree/
TestBinaryTree	Tests	/BinaryTree/
TestHeap	TestHeap	/Heap/
TestHeap	TestWrapper	/Heap/
TestHeap	Tester	/Heap/
TestHeap	Tests	/Heap/
TestDictionary	TestWrapper	/Dictionary/
TestDictionary	Tester	/Dictionary/
TestDictionary	Tests	/Dictionary/
	Program.cs (file)	/